

Simplifying Negative Goals Using Typed Existence Properties

Lunjin Lu

Department of Computer Science and Engineering, Oakland University

John G. Cleary

Department of Computer Science, University of Waikato

Abstract

A method for extracting positive information from negative goals is proposed. It makes use of typed existence properties between arguments of a predicate to rewrite negative goals in a logic program. A typed existence property is a generalization of functional dependencies in that an input value maps to a fixed number of output values. Types are used to specify the domains of the input and output values. An implementation of the simplification method is presented and its complexity is analyzed. A key algorithm of the implementation checks if an atom in a negative goal can be extracted using a given typed existence property. A digraph links an atom to the quantified variables occurring in the atom and is used to quickly retrieve atoms in the negative goal that may become extractable after some other atom is extracted.

Key words: Constructive negation, Logic Programs, Existence properties, Types

1 Introduction

A challenging issue in logic programming is how to find answers to negative goals. Chan introduced the “constructive negation” rule which allows non-ground negative goals to bind variables in the same way as positive ones [7,8]. Many methods along this line have been proposed [2,4,5,7,8,13,14,18,19,22,23,24,27,28,29]. These methods find answers to negative goals by negating a frontier of a derivation tree for the negated sub-goal.

Email addresses: lunjin@acm.org (Lunjin Lu), jcleary@cs.waikato.nz (John G. Cleary).

A different approach was proposed by Cleary that makes use of *existence* properties of arithmetic constraints to rewrite negative goals [9]. There are usually functional dependencies between arguments to an arithmetic constraint. Let $add(x, y, z)$ denote $z = x + y$ on the domain of integers, for any integers x and y , then there is a unique z such that $add(x, y, z)$ is true. This is called an *exists unique* property. It implies that $\neg\exists z.add(x, y, z)$ is unsatisfiable and that $\neg\exists z.(add(x, y, z) \wedge q(z))$ can be directly simplified to $add(x, y, z) \wedge \neg q(z)$. Another kind of property is called the *exists sometimes* property. Let $log(y, x)$ denote $y = 10^x$ on the domain of integers. Then there is at most one x such that $log(y, x)$ is true. So, we can directly simplify $\neg\exists x.(log(y, x) \wedge q(x))$ to $\neg\exists x.log(y, x) \vee log(y, x) \wedge \neg q(x)$. The simplification procedure in [9] consists of rewrite rules for these kinds of property.

The prerequisite that a functional or partial functional dependency exists between arguments to a predicate (arithmetic constraints in [9]) is over restrictive. Consider $sq(x, y)$ in the domain of real numbers where $sq(x, y)$ denotes $y = x^2$. For every x , there is a unique y such that $sq(x, y)$ is true. However, for every $y > 0$, there are two x 's such that $sq(x, y)$ is true. The rewrite rule for *exists unique* properties in [9] doesn't apply directly when it comes to simplifying $\neg\exists x.(sq(x, y) \wedge b(x))$. This problem is resolved by inserting a tautology $(x \geq 0 \vee x < 0)$ into the negative goal and transforming $\neg\exists x.(sq(x, y) \wedge b(x))$ into $\neg\exists x_1.(sq(x_1, y) \wedge x_1 \geq 0 \wedge b(x_1)) \wedge \neg\exists x_2.(sq(x_2, y) \wedge x_2 < 0 \wedge b(x_2))$ and then applying the rewrite rule for *exists unique* properties to the two negative sub-goals. This causes difficulty because we need to have *exists unique* properties for complex constraints $(sq(x_1, y) \wedge x_1 \geq 0)$ and $(sq(x_2, y) \wedge x_2 < 0)$. Moreover, inserting a correct tautology, say $(x \geq 0 \vee x < 0)$, into the negative goal before rewriting is rather involved and difficult to mechanise.

This paper generalizes the simplification method in [9] and presents an heuristic implementation of the generalized method. An input may now correspond to multiple outputs provided that each output can be isolated into a sub-domain that is expressed as a type. The generalized method is applicable to more negative goals because use of types admits more *existence* properties and therefore allows more negative goals to be rewritten. The simplification method can be applied in program transformation because it extracts an atom from a negative goal without executing the atom.

A crucial task of any implementation of the generalized method is to introduce new local (i.e., existentially quantified) variables into an atom inside a negative goal so that it satisfies a given *existence* property. Consider $\neg(sq(x, 16) \wedge q(16))$ and this *existence* property.

For any real number x , there is a unique real number y such that $sq(x, y)$ is true. (P)

The atom $sq(x, 16)$ doesn't satisfy property (P) and hence cannot be extracted. This is because the unique y such that $sq(x, y)$ holds is not necessarily 16. However, $\neg(sq(x, 16) \wedge q(16))$ can be transformed to $\neg\exists y'.(sq(x, y') \wedge 16 = y' \wedge q(16))$ by introducing a new local variable y' . The transformed goal can then be rewritten to $sq(x, y') \wedge \neg(16 = y' \wedge q(16))$ since $sq(x, y')$ satisfies property (P). In general cases, the task of introducing new local variables is much more complicated. We present an algorithm that tests if an *existence* property can be used to extract an atom by introducing zero or more new local variables.

Another essential issue is how to find quickly an extractable atom inside a negative goal. Let G_i be $sq(x_{i-1}, x_i)$ and G be $\neg\exists x_1.\exists x_2.\dots\exists x_n.\exists x_{n+1}.[G_n \wedge G_{n-1} \dots G_2 \wedge G_1]$. By repeatedly using property (P), we can extract from G atoms G_1, G_2 to G_{n-1} in order and obtain $G_1 \wedge G_2 \dots G_{n-1} \wedge \neg\exists x_{n+1}.G_n$. Observe that G_j becomes extractable after and only after x_j becomes global upon extraction of G_{j-1} . We use a digraph to represent a negative goal. The digraph links an atom to a local variable iff the local variable occurs in the atom. This data structure allows efficient identification of extractable atoms.

The rest of the paper is organized as follows. Section 2 presents the generalized simplification method. Section 3 describes digraphs for representing negative goals and section 4 presents the algorithm for introducing new variables. Section 5 describes briefly the implementation in ECLipSe Prolog and section 6 analyzes its complexity. Section 7 discusses related work and section 8 concludes. A preliminary version of this paper appeared in Proceedings of ACM SAC'07, March 11-14, 2007 Seoul, Korea except section 2 that is a major revision of [10].

1.1 Notations

We assume that negative goals are of the form $\neg\exists L.G$ where L is a set of variables and G a conjunction of atoms. We also assume that variables are typed. Expression $y:\eta$ indicates that variable y has type η . A type is a finite expression denoting a possibly infinite set of terms. We use $\mathbf{1}$ to denote the set of all ground terms and $\mathbf{0}$ the empty set of terms. Types \mathcal{R} and \mathcal{Z} denote the set of real numbers and the set of integer numbers respectively. Types \mathcal{R} and \mathcal{Z} with subscripts denote their subtypes. A subscript is either an interval or a logical formula. For instance, $\mathcal{Z}_{<0}$ denotes the set of negative integers and $\mathcal{R}_{[0,1]}$ the real interval $[0, 1)$. Relation $\sigma \sqsubseteq \theta$ holds iff σ is a subtype of θ ; and relation $\sigma \equiv \theta$ holds iff σ is equivalent to θ . The intersection of two types θ and σ is denoted as $\theta \sqcap \sigma$. We forgo the presentation of a type system because any type system for logic programs such as [15,20,30] can be used. We also assume that a set of typed existence properties are given.

Both existence properties and rewrite rules partition the argument list of an atom into several vectors. For an example, let $\text{add}(x,y,z)$ denote $x + y = z$ where x, y and z range over the domain of real numbers. For given x and y , there is exactly one z such that $\text{add}(x,y,z)$ holds. The input vector π_i consists of the first two arguments x and y and the output vector π_o consists of the third argument z . Formally, a vector is a partial function whose domain is a set of argument positions (positive integers). Thus, $\pi_i = \{1 \mapsto x, 2 \mapsto y\}$ and $\pi_o = \{3 \mapsto z\}$. The domain of a vector π is denoted $\text{dom}(\pi)$. The projection of π onto $D \subseteq \text{dom}(\pi)$ is denoted $\pi \downarrow D$. Then $(\pi \downarrow D)(i) = \pi(i)$ if $i \in D$. Otherwise, $(\pi \downarrow D)(i)$ is undefined. We call $\pi \downarrow D$ a sub-vector of π and accordingly π is a super-vector of $\pi \downarrow D$. The empty vector is denoted by ϵ . We have $\pi \downarrow \emptyset = \epsilon$ for any vector π . By an element of a vector π , we mean $\pi(i)$ for some $i \in \text{dom}(\pi)$. We use $\text{diff}(\pi)$ to indicate that elements in π are pair wise different, i.e., $\text{diff}(\pi)$ is true iff $\pi(p_1) \neq \pi(p_2)$ for any $p_1 \in \text{dom}(\pi)$ and any $p_2 \in \text{dom}(\pi)$ such that $p_1 \neq p_2$. In the sequel, a letter with an over bar \bar{u} denotes a vector of different variables, a letter with a tilde \tilde{u} denotes a vector of terms and a Greek letter with an over bar $\bar{\eta}$ denotes a vector of types. A vector of types is also called a type. When there is no ambiguity from the context, \bar{u} is also used to denote the set of variables occurring in \bar{u} . For instance, put $\bar{x} = \{1 \mapsto x_1, 2 \mapsto x_2\}$, we write $\exists \bar{x}.p(\bar{x})$ instead of $\exists x_1.\exists x_2.p(\bar{x})$. By juxtaposition $\pi_1\pi_2$, we mean that π_1 and π_2 have disjoint domains and $\pi_1\pi_2 = \pi_1 \cup \pi_2$. For instance, $\pi_i\pi_o = \pi_o\pi_i = \{1 \mapsto x, 2 \mapsto y, 3 \mapsto z\}$. Let p be of arity n . By $p(\pi)$, we mean that $\text{dom}(\pi) = \{1..n\}$ and $p(\pi) = p(\pi(1), \dots, \pi(n))$. For instance, $\text{add}(\pi_i\pi_o)$ stands for $\text{add}(x, y, z)$. When it is clear from context, a vector is simply written as a sequence with positions omitted.

By $\bar{u} : \bar{\sigma}$, we mean that $\text{dom}(\bar{u}) = \text{dom}(\bar{\sigma})$ and $\bar{u}(i) : \bar{\sigma}(i)$ for all $i \in \text{dom}(\bar{u})$. By $\bar{\sigma} \sqsubseteq \bar{\eta}$, we mean that $\text{dom}(\bar{\eta}) = \text{dom}(\bar{\sigma})$ and $\bar{\sigma}(i) \sqsubseteq \bar{\eta}(i)$ for all $i \in \text{dom}(\bar{\sigma})$. We say that $\bar{\sigma}$ and $\bar{\eta}$ intersect iff $\bar{\sigma}(i) \cap \bar{\eta}(i) \neq \mathbf{0}$ for all $i \in \text{dom}(\bar{\sigma})$. Let E be an expression. We use \mathbf{V}_E to denote the set of variables in E and $\text{type}(E)$ the type of E .

2 Generalized Method

This section generalizes the simplification method in [9]. We first generalize the notion of an *existence* property and then the rewrite rules that make use of *existence* properties.

One rewrite rule applies when it is known that for every input value a predicate holds for exactly one output value. Another applies when it is known that for every input value a predicate holds for at most one output value. It is not necessary to have the output value available in order to apply these two rewrite rules. What these two rewrite rules make use of is knowledge of whether for

every input value a predicate holds for exactly one output value or for at most one output value.

2.1 Typed Existence Properties

An *exists unique* property in [9] expresses that, for every \bar{u} , there is exactly one \bar{x} such that $p(\bar{u}\bar{x})$ holds. In other words, predicate “p” is be a function from the domain of \bar{u} to that of \bar{x} . Parameters in \bar{u} and \bar{x} can be viewed respectively as input and output parameters. The predicate “p” may satisfy more than one *exists unique* properties with different groups of input and output parameters.

As mentioned in section 1, functional dependency is a strong requirement of a predicate in that many interesting properties cannot be expressed as functional dependencies. For an instance, let $exp(x, y)$ denote $y = 10^x$ over the domain of real numbers. Then exp is not a total function from y to x since there is no x such that $exp(x, y)$ holds for any $y < 0$. This problem can be resolved by restricting an input to a sub-domain of its domain. For instance, the property that for every $y > 0$ there is exactly one x such that $exp(x, y)$ holds can be expressed as $\forall y : \mathcal{R}_{>0}. \exists! x : \mathcal{R}. exp(x, y)$ where $\exists!$ means “there is exactly one”. Types also admits more precise properties. For instance, the property that for any real number x there is exactly one non-negative real number y such that $exp(x, y)$ holds can be expressed as $\forall x : \mathcal{R}. \exists! y : \mathcal{R}_{\geq 0}. exp(x, y)$. Another way to generalize the notion of an *exists unique* property is to allow an input value to correspond to more than one output value. A typed *exists unique* property of a predicate thus expresses that for every input value of a given sub-domain, the predicate holds for a fixed number of output values each of which can be isolated into a sub-domain. For instance, each positive number has two square roots one of which is positive and the other is negative. Formally, a typed *exists unique* property has the following form where I is a finite set of indices.

$$\forall \bar{u} : \bar{\sigma}. \forall \bar{x}. [p(\bar{u}\bar{x}) \rightarrow \bigvee_{i \in I} \bar{x} \in \bar{\theta}_i] \quad (1)$$

$$\forall \bar{u} : \bar{\sigma}. \bigwedge_{i \in I} \exists! \bar{x}_i : \bar{\theta}_i. p(\bar{u}\bar{x}_i) \quad (2)$$

Each $\bar{\theta}_i$ is called an *output subtype* of the output parameter \bar{x} . Note that the type of an input parameter expresses the condition under which a specific property holds.

Example 1 *The fact that, in the domain of real numbers, a positive number has exactly one negative square root and exactly one positive square root can be expressed as the following exists unique property.*

$$\forall y : \mathcal{R}_{>0}. \forall x. (sq(x, y) \rightarrow x \in \mathcal{R}_{>0} \vee x \in \mathcal{R}_{<0})$$

$$\forall y:\mathcal{R}_{>0}.(\exists!x_1:\mathcal{R}_{>0}.sq(x_1, y) \wedge \exists!x_2:\mathcal{R}_{<0}.sq(x_2, y))$$

Example 2 *The fact that the square of any real number is a positive real number is expressed as follows.*

$$\begin{aligned} \forall x:\mathcal{R}.\forall y.(sq(x, y) \rightarrow y \in \mathcal{R}_{\geq 0}) \\ \forall x:\mathcal{R}.\exists!y:\mathcal{R}_{\geq 0}.sq(x, y) \end{aligned}$$

Note that we have restricted the domain of y to $\mathcal{R}_{\geq 0}$ rather than \mathcal{R} , which helps avoid the introduction of local variables in some cases as explained later.

An *exists sometimes* properties is generalized in the same way, so that every input value has at most one output value in each of a fixed number of sub-domains. Formally, a typed *exists sometimes* property is expressed by (1) and

$$\forall \bar{u}:\bar{\sigma}.\wedge_{i \in I} \exists? \bar{x}_i:\bar{\theta}_i.p(\bar{u}\bar{x}_i) \quad (3)$$

where $\exists?$ denotes “there is at most one”. Formula (3) requires that, for each \bar{u} of type $\bar{\sigma}$, there is at most one \bar{x} in each $\bar{\theta}_i$ such that $p(\bar{u}\bar{x})$ holds. An example of typed *exists sometimes* properties can be found in Ex. 13.

A typed *exists* property $\forall \bar{u}:\bar{\sigma}.\exists \bar{x}:\bar{\theta}.p(\bar{u}\bar{x})$ states that for every \bar{u} of type $\bar{\sigma}$ there are some \bar{x} of type $\bar{\theta}$ such that $p(\bar{u}\bar{x})$ holds. For instance, the *append/3* program satisfies $\forall z:\text{list}(\beta).\exists x:\text{list}(\beta).\exists y:\text{list}(\beta).append(x, y, z)$ which states that every list z can be split into two lists x and y .

A typed miscellaneous property $\forall \bar{u}:\bar{\sigma}.(p(\bar{u}) \leftrightarrow q(\bar{u}))$ states that, for every \bar{u} of type $\bar{\sigma}$, $p(\bar{u})$ can be replaced by $q(\bar{u})$. For instance, we have $\forall x:\mathcal{Z}.y:\mathcal{Z}.\neg(x < y) \leftrightarrow (x \geq y)$.

2.2 Rewrite Rule for Exists Unique Properties

We now derive a rewrite rule that make uses of typed *existence* properties. Consider first typed *exists unique* properties. From (1), we have $p(\tilde{u}\tilde{x}) \leftrightarrow (p(\tilde{u}\tilde{x}) \wedge [\forall_{i \in I} \tilde{x} \in \bar{\theta}_i])$. Hence $(p(\tilde{u}\tilde{x}) \wedge Q) \leftrightarrow (\forall_{i \in I} p(\tilde{u}\tilde{x}) \wedge (\tilde{x} \in \bar{\theta}_i) \wedge Q)$. Distributing \exists over \forall , renaming local variables within their scopes and applying De Morgan’s law, we obtain

$$\neg \exists \bar{x}\bar{y}.[p(\tilde{u}\tilde{x}) \wedge Q] \leftrightarrow \wedge_{i \in I} \neg \exists \bar{x}_i\bar{y}.[p(\tilde{u}\tilde{x}_i) \wedge (\bar{x}_i \in \bar{\theta}_i) \wedge Q[\bar{x}/\bar{x}_i]]$$

provided that $\mathbf{V}_{\bar{u}} \cap (\bar{x} \cup \bar{y}) = \emptyset$ holds where $Q[\bar{x}/\bar{x}_i]$ is the result of substituting \bar{x}_i for \bar{x} in Q . Note that \bar{x} is renamed into \bar{x}_i for each output subtype $\bar{\theta}_i$.

The condition $\mathbf{V}_{\tilde{u}} \cap (\bar{x} \cup \bar{y}) = \emptyset$ ensures that \tilde{u} does not contain local variables. To see why this is necessary, assume the *exists unique* property for integer addition in the introduction, $\neg \exists y : \mathcal{Z}.(\text{add}(x, y, y) \wedge q(y))$ cannot be simplified to $\text{add}(x, y, y) \wedge \neg q(y)$ because $\neg \exists y : \mathcal{Z}.\text{add}(x, y, y)$ holds for $x \neq 0$. The fact that the second argument y to *add* is a local variable invalidates the condition.

For $p(\tilde{u}\bar{x})$ to be extracted, its output arguments must satisfy this requirement.

An output argument is a local variable; and for each output subtype T_p of its corresponding output parameter, either T_p is a subtype of T_a or T_p doesn't intersect with T_a where T_a is the type of the output argument.

Example 3 *This is an exists unique property in the domain of integers.*

$$\begin{aligned} \forall x:\mathcal{Z}.\forall y:\mathcal{Z}.\forall z.(\text{add}(x, y, z) \rightarrow z \in \mathcal{Z}) \\ \forall x:\mathcal{Z}.\forall y:\mathcal{Z}.\exists! z:\mathcal{Z}.\text{add}(x, y, z) \end{aligned}$$

It states that, for any integers x and y , there is a unique integer z such that $\text{add}(x, y, z)$ is true. It would be wrong to use the property to rewrite $\neg \exists z : \mathcal{Z}_{[-\infty, 10]}.(\text{add}(10, y, \mathcal{Z}, z) \wedge b(z))$ into $\text{add}(10, y, \mathcal{Z}, z:\mathcal{Z}_{[-\infty, 10]}) \wedge \neg b(z)$. This is because z can take any value in \mathcal{Z} and $\mathcal{Z}_{[-\infty, 10]}$ is not a supertype of \mathcal{Z} .

The number of solutions to be negated is limited by the number of output subtypes of the output parameter. Some output subtypes are not relevant for a particular negative goal. An output subtype is relevant iff it intersects with the type of the local variable in the negative goal. We call an index a *relevant index* if its corresponding output subtype is relevant. We only need to consider relevant output subtypes when rewriting the negative goal.

Example 4 *Let G be $\neg \exists x:\mathcal{R}_{\geq 0}.(\text{sq}(x, y:\mathcal{R}_{>0}) \wedge b(x))$. From Ex. 1, $\text{sq}(x, y:\mathcal{R}_{>0})$ has two solutions for x , one of them is in $\mathcal{R}_{<0}$ and the other is in $\mathcal{R}_{>0}$. This suggests that there are two solutions to be negated. But, the type $\mathcal{R}_{\geq 0}$ of the local variable x doesn't intersect with $\mathcal{R}_{<0}$, i.e., only output subtype $\mathcal{R}_{>0}$ is relevant for G . G is rewritten to $\text{sq}(x_1:\mathcal{R}_{>0}, y:\mathcal{R}_{>0}) \wedge \neg b(x_1)$ since the type $\mathcal{R}_{\geq 0}$ of x is a supertype of the relevant output subtype $\mathcal{R}_{>0}$.*

The following rewrite rule makes uses of typed *exists* properties. It verifies that an input argument is of the type of the corresponding input parameter and that the type of an output argument is a supertype of the type of the corresponding output parameter.

ET
Given $\forall \bar{u}:\bar{\sigma}.\exists \bar{x}:\bar{\theta}.p(\bar{u}\bar{x})$ and $\text{type}(\tilde{u}) \sqsubseteq \bar{\sigma} \wedge \mathbf{V}_{\tilde{u}} \cap \bar{x} = \emptyset \wedge \bar{\theta} \sqsubseteq \bar{\eta}$
$\neg \exists \bar{x}:\bar{\eta}.p(\tilde{u}\bar{x}) \leftrightarrow \text{false}$

The following miscellaneous rewrite rule verifies that an input argument is of the type of the corresponding input parameter.

RT
Given $\forall \bar{u}:\bar{\sigma}.(\neg p(\bar{u}) \leftrightarrow q(\bar{u}))$ and $type(\tilde{u}) \sqsubseteq \bar{\sigma}$
$\neg p(\tilde{u}) \leftrightarrow q(\tilde{u})$

When the requirement on output arguments of an atom is not met, new local variables need be introduced so that the atom can be extracted. Consider how an *exists unique* property can be used to rewrite negative goals of the form

$$\neg \exists \mathbf{L}. [p(\tilde{u}\tilde{x}) \wedge Q] \quad (g1)$$

where \mathbf{L} is a set of typed variables. Assume that \tilde{u} is of type $\bar{\sigma}$ (I.e. $type(\tilde{u}) \sqsubseteq \bar{\sigma}$) and that variables in \mathbf{L} do not occur in \tilde{u} (I.e. $\mathbf{V}_{\tilde{u}} \cap \mathbf{L} = \emptyset$). Then $\exists \bar{x}. (p(\tilde{u}\bar{x}) \wedge (\bar{x} = \tilde{x}) \wedge Q)$ is equivalent to $\vee_{i \in I} \exists \bar{x}_i. (p(\tilde{u}\bar{x}_i : \bar{\theta}_i) \wedge (\bar{x}_i = \tilde{x}) \wedge Q)$ from (1). Goal (g1) is equivalent to $\neg \exists \mathbf{L}. \exists \bar{x}. [p(\tilde{u}\bar{x}) \wedge (\bar{x} = \tilde{x}) \wedge Q]$ and hence is equivalent to $\neg \exists \mathbf{L}. [\vee_{i \in I} \exists \bar{x}_i. (p(\tilde{u}\bar{x}_i : \bar{\theta}_i) \wedge (\bar{x}_i = \tilde{x}) \wedge Q)]$. Distributing \exists over \vee , applying De Morgan's law and using (2), we deduce that goal (g1) is equivalent to

$$\wedge_{i \in I} [p(\tilde{u}\bar{x}_i : \bar{\theta}_i) \wedge \neg \exists \mathbf{L}. ((\tilde{x} = \bar{x}_i) \wedge Q)] \quad (g2)$$

provided that (1), (2), $type(\tilde{u}) \sqsubseteq \bar{\sigma}$ and $\mathbf{V}_{\tilde{u}} \cap \mathbf{L} = \emptyset$ hold.

Example 5 Let the exists unique property be that in Ex. 1 and the negative goal be

$$\neg \exists z':\mathcal{Z}, x':\mathcal{R}_{\geq 20}. (sq(x', y':\mathcal{R}_{>10}) \wedge Q(x', z')) \quad (g1')$$

Goal (g1') is an instance of (g1). We have $\mathbf{L} = \{z':\mathcal{Z}, x':\mathcal{R}_{\geq 20}\}$, $\tilde{u} = y'$ and $\tilde{x} = x'$. It holds that $y' \in \mathcal{R}_{>0}$ since $y' \in \mathcal{R}_{>10}$ and $(\mathcal{R}_{>10} \sqsubseteq \mathcal{R}_{>0})$. It also holds that $\mathbf{V}_{\tilde{u}} \cap \mathbf{L} = \{y'\} \cap \{z', x'\} = \emptyset$. Therefore, (g1') rewrites to

$$\left(\begin{array}{l} sq(x_1:\mathcal{R}_{>0}, y':\mathcal{R}_{>10}) \wedge \neg \exists z':\mathcal{Z}, x':\mathcal{R}_{\geq 20}. (x' = x_1 \wedge Q(x', z')) \\ \wedge \quad sq(x_2:\mathcal{R}_{<0}, y':\mathcal{R}_{>10}) \wedge \neg \exists z':\mathcal{Z}, x':\mathcal{R}_{\geq 20}. (x' = x_2 \wedge Q(x', z')) \end{array} \right) \quad (g2')$$

If $type(\tilde{x})$ doesn't intersects with $\bar{\theta}_k$ then $p(\tilde{u}\bar{x}_k : \bar{\theta}_k) \wedge \neg \exists \mathbf{L}. ((\tilde{x} = \bar{x}_k) \wedge Q)$ can be removed from (g2) because $(\tilde{x} = \bar{x}_k)$ is unsatisfiable and any further instantiation of \bar{x}_k has no effect on the variables of the original goal. Let \mathbf{W} be the set of those elements of \mathbf{L} that occur in \tilde{x} and $\mathbf{Y} = \mathbf{L} \setminus \mathbf{W}$. Then

$\neg\exists L.((\tilde{x} = \bar{x}_j) \wedge Q)$ is equivalent to

$$\neg\exists W_j.(\tilde{x}[W/W_j] = \bar{x}_j) \vee (\tilde{x}[W/W_j] = \bar{x}_j) \wedge \neg\exists Y.Q[W/W_j] \quad (g3)$$

where W_j is a renaming of W . The disequality constraint $\neg\exists W_j.(\tilde{x}[W/W_j] = \bar{x}_j)$ can be dealt with by augmenting Chan's simplification procedure with types.

Example 6 *Continue with Ex. 5. We have $W = \{x':\mathcal{R}_{\geq 20}\}$, $Y = \{z':\mathcal{Z}\}$ and $J = \{1\}$. The output subtype $\mathcal{R}_{<0}$ is not relevant since $(\text{type}(x') \sqcap \mathcal{R}_{<0}) \equiv \mathbf{0}$. The sub-formula $\neg\exists z':\mathcal{Z}, x':\mathcal{R}_{\geq 20}.(x' = x_1 \wedge Q(x', z'))$ in (g2') can be rewritten to $\neg\exists w_1:\mathcal{R}_{\geq 20}.(w_1 = x_1) \vee (w_1:\mathcal{R}_{\geq 20} = x_1) \wedge \neg\exists z':\mathcal{Z}.Q(w_1, z')$.*

A new local variable is introduced for each output argument in (g3). As the cost of simplifying $\neg\exists W_j.(\tilde{x}[W/W_j] = \bar{x}_j)$ increases with the number of equations it contains, it is desirable to avoid introducing new local variables whenever possible. No new local variable need be introduced for an output argument r if r is a local variable, its type is a super-type of all relevant output subtypes and it doesn't appear in any other output argument.

Example 7 *Continue with Ex. 5. Variable x' is a local variable. Its type is $\mathcal{R}_{\geq 20}$. The only relevant output subtype is $\mathcal{R}_{>0}$. A new local variable was introduced because $\mathcal{R}_{\geq 20}$ is not a super-type of $\mathcal{R}_{>0}$.*

Example 8 *The following is an exists unique property in the domain of integers.*

$$\begin{aligned} \forall x:\mathcal{Z}.y:\mathcal{Z}.\forall z.(add(x, y, z) \rightarrow z \in \mathcal{Z}) \\ \forall x:\mathcal{Z}.y:\mathcal{Z}.\exists! z:\mathcal{Z}.add(x, y, z) \end{aligned}$$

It states that, for any integers x and y , there is a unique integer z such that $add(x, y, z)$ is true. It would be wrong to use the exists unique property to rewrite $\neg \exists z:\mathcal{Z}_{[-\infty, 10]}.(add(10, y, z) \wedge b(z))$ into $add(10, y, z:\mathcal{Z}_{[-\infty, 10]}) \wedge \neg b(z)$. This is because z can take any value in \mathcal{Z} and $\mathcal{Z}_{[-\infty, 10]}$ is not a super-type of \mathcal{Z} .

Example 9 *The fact that, in the domain of real numbers, a positive number has exactly one negative square root and exactly one positive square root can be expressed as the following exists unique property.*

$$\begin{aligned} \forall y:\mathcal{R}_{>0}.\forall x.(sq(x, y) \rightarrow x \in \mathcal{R}_{>0} \vee x \in \mathcal{R}_{<0}) \\ \forall y:\mathcal{R}_{>0}.(\exists! x_1:\mathcal{R}_{>0}.sq(x_1, y) \wedge \exists! x_2:\mathcal{R}_{<0}.sq(x_2, y)) \end{aligned}$$

Let the negative goal to rewrite be the following.

$$\neg\exists x:\mathcal{R}_{\geq 0}.(sq(x, y:\mathcal{R}_{>0}) \wedge b(x))$$

QVT
Given (1), (2), $type(\tilde{u}) \sqsubseteq \bar{\sigma}$ and $\mathbf{V}_{\tilde{u}} \cap \mathbf{L} = \emptyset$ hold
$\neg \exists \mathbf{L}. [p(\tilde{u}\tilde{x}) \wedge Q] \leftrightarrow$ $\left(\begin{array}{l} \text{let } J = \{i \in I \mid type(\tilde{x}) \cap \bar{\theta}_i \neq \mathbf{0}\} \\ \nu \subseteq \{p \mid p \in dom(\tilde{x}) \wedge \tilde{x}(p) \in \mathbf{L} \wedge \forall j \in J. (\bar{\theta}_j(p) \sqsubseteq type(\tilde{x}(p)))\} \\ \text{such that } diff(\tilde{x} \downarrow \nu) \text{ holds} \\ \mu = dom(\tilde{x}) \setminus \nu, \quad \bar{r} = \tilde{x} \downarrow \nu, \quad \tilde{s} = \tilde{x} \downarrow \mu \\ \mathbf{W} = (\mathbf{L} \cap \mathbf{V}_{\tilde{s}}) \setminus \bar{r}, \quad \mathbf{Y} = \mathbf{L} \setminus \mathbf{W} \\ \text{in} \\ \bigwedge_{j \in J} \left(\begin{array}{l} p(\tilde{u}[\bar{z}_j \bar{r}_j]:\bar{\theta}_j) \wedge \neg \exists \mathbf{W}_j. (\tilde{s}[\bar{r}/\bar{r}_j, \mathbf{W}/\mathbf{W}_j] = \bar{z}_j) \\ \vee \quad p(\tilde{u}[\bar{z}_j \bar{r}_j]:\bar{\theta}_j) \wedge (\tilde{s}[\bar{r}/\bar{r}_j, \mathbf{W}/\mathbf{W}_j] = \bar{z}_j) \wedge \neg \exists \mathbf{Y}. Q[\bar{r}/\bar{r}_j, \mathbf{W}/\mathbf{W}_j] \end{array} \right) \end{array} \right)$

Fig. 1. Rewrite rule QVT for *exists unique* properties.

where the type of a variable is associated with its first occurrence. By the above exists unique property, $sq(x, y:\mathcal{R}_{>0})$ has two solutions for x , one of them is in $\mathcal{R}_{<0}$ and the other is in $\mathcal{R}_{>0}$. This suggests that there are two solutions to be negated. But, the type $\mathcal{R}_{\geq 0}$ of the local variable x doesn't intersect with $\mathcal{R}_{<0}$, that is, only output subtype $\mathcal{R}_{>0}$ is relevant for the negative goal. The negative goal is rewritten to

$$sq(x_1:\mathcal{R}_{>0}, y:\mathcal{R}_{>0}) \wedge \neg b(x_1)$$

since the type $\mathcal{R}_{\geq 0}$ of x is a super-type of the relevant output subtype $\mathcal{R}_{>0}$.

The above considerations lead to the rewrite rule QVT for *exists unique* properties in Fig. 1. The condition $type(\tilde{u}) \sqsubseteq \bar{\sigma} \wedge \mathbf{V}_{\tilde{u}} \cap (\bar{x} \cup \bar{y}) = \emptyset$ in the rewrite rule ensures that an input argument is of the type of its corresponding input parameter and it doesn't contain any local variables. QVT generates only sub-formulae for relevant output subtypes which are collected by $J = \{i \in I \mid (\bar{\eta} \cap \bar{\theta}_i) \neq \mathbf{0}\}$. Variables in $\bar{z}_j \bar{r}_j$ and \mathbf{W}_j do not occur in the left hand side of the rewrite rule. The vector $\bar{z}_j \bar{r}_j$ is typed with $\bar{\theta}_j$ while \mathbf{W}_j inherits the type of \mathbf{W} . The vector \bar{r} consists of different variables; and it is a sub-vector of \tilde{x} for which no new local variables need be introduced.

Example 10 Continue with Ex. 1 and Ex. 5. QVT rewrites $(g1')$ directly to

$$sq(x_1:\mathcal{R}_{>0}, y':\mathcal{R}_{>10}) \wedge \neg \exists w_1:\mathcal{R}_{\geq 20}. (w_1 = x_1)$$

$$\vee \quad sq(x_1:\mathcal{R}_{>0}, y':\mathcal{R}_{>10}) \wedge (w_1:\mathcal{R}_{\geq 20} = x_1) \wedge \neg \exists z':\mathcal{Z}. Q(w_1, z')$$

Example 11 The `append/3` program satisfies this exists unique property.

$$\begin{aligned} & \forall x:\text{list}(\beta), y:\text{list}(\beta).z.(append(x, y, z) \rightarrow z:\text{list}(\beta)) \\ & \forall x:\text{list}(\beta), y:\text{list}(\beta)\exists!z:\text{list}(\beta).append(x, y, z) \end{aligned}$$

Goal $\neg\exists z:\text{list}(\beta).(append(x:\text{list}(\beta), y:\text{list}(\beta), z), p(z))$ is rewritten to $append(x:\text{list}(\beta), y:\text{list}(\beta), z:\text{list}(\beta)), \neg p(z)$ by QVT.

When QVT is used as a simplification rule, it will prune unsatisfiable goals without doing a satisfiability test.

Example 12 We have $\forall y:\mathbf{1}.x.(x = s(y) \rightarrow x:\mathbf{1})$ and $\forall y:\mathbf{1}.\exists!x:\mathbf{1}.(x = s(y))$ in the domain of Herbrand universe. Consider the following program.

$p(y).$
 $r(y) :- x=s(y), q(x).$

The goal $p(y:\mathbf{1}), \neg r(y)$ is reduced to $p(y), \neg\exists x:\mathbf{1}.(x = s(y:\mathbf{1}), q(x))$ which is then simplified directly into $x:\mathbf{1} = s(y), p(y:\mathbf{1}), \neg q(x)$ using the above property. Without using this property, $\neg\exists x:\mathbf{1}.(x = s(y:\mathbf{1}), q(x))$ is simplified to

$$\forall x:\mathbf{1}.(x \neq s(y:\mathbf{1})) \vee (x:\mathbf{1} = s(y:\mathbf{1}), \neg q(x))$$

and a satisfiability test is then used to eliminate $\forall x:\mathbf{1}.(x \neq s(y:\mathbf{1}))$. In that sense, the satisfiability test is pushed into the simplification procedure by the exists unique property.

2.3 Rewrite Rule for Exists Sometimes Properties

The same considerations as in the case for *exists unique* properties lead to the rewrite rule SVT for *exists sometimes* properties in Fig. 2.

Example 13 The fact that, in the domain of integer numbers, a positive number has at most one negative square root and at most one positive square root can be expressed as the following typed exists sometimes property.

$$\begin{aligned} & \forall y:\mathcal{Z}_{>0}.\forall x.(sq(x, y) \rightarrow x \in \mathcal{Z}_{<0} \vee x \in \mathcal{Z}_{>0}) \\ & \forall y:\mathcal{Z}_{>0}.(\exists?x_1:\mathcal{Z}_{<0}.sq(x_1, y) \wedge \exists?x_2:\mathcal{Z}_{>0}.sq(x_2, y)) \end{aligned}$$

The local variable x in the negative goal $\neg\exists x:\mathcal{Z}_{[0,20]}.(sq(x, y:\mathcal{Z}_{>0}) \wedge b(x))$ has a type $\mathcal{Z}_{[0,20]}$ which is not a super-type of the sole relevant output subtype $\mathcal{Z}_{>0}$ of the corresponding output parameter. Therefore, a new local variable z_2 of type $\mathcal{Z}_{>0}$ is introduced and the negative goal is rewritten to the following.

$$\neg\exists z_2:\mathcal{Z}_{>0}.sq(z_2, y:\mathcal{Z}_{>0})$$

SVT
Given (1), (3), $type(\tilde{u}) \sqsubseteq \bar{\sigma}$ and $\mathbf{V}_{\tilde{u}} \cap \mathbf{L} = \emptyset$ hold
$\neg \exists \mathbf{L}. [p(\tilde{u}\tilde{x}) \wedge Q] \leftrightarrow$ $\left(\begin{array}{l} \text{let } J = \{i \in I \mid type(\tilde{x}) \sqcap \bar{\theta}_i \neq \mathbf{0}\} \\ \nu \subseteq \{p \mid p \in dom(\tilde{x}) \wedge \tilde{x}(p) \in \mathbf{L} \wedge \forall j \in J. (\bar{\theta}_j(p) \sqsubseteq type(\tilde{x}(p)))\} \\ \text{such that } diff(\tilde{x} \downarrow \nu) \text{ holds} \\ \mu = dom(\tilde{x}) \setminus \nu, \quad \bar{r} = \tilde{x} \downarrow \nu, \quad \tilde{s} = \tilde{x} \downarrow \mu \\ \mathbf{W} = (\mathbf{L} \cap \mathbf{V}_{\tilde{s}}) \setminus \bar{r}, \quad \mathbf{Y} = \mathbf{L} \setminus \mathbf{W} \\ \text{in} \\ \bigwedge_{j \in J} \left(\begin{array}{l} \neg \exists (\bar{z}_j \bar{r}_j) : \bar{\theta}_j. p(\tilde{u} \bar{z}_j \bar{r}_j) \\ \vee p(\tilde{u}[\bar{z}_j \bar{r}_j] : \bar{\theta}_j) \wedge \neg \exists \mathbf{W}_j. (\tilde{s}[\bar{r}/\bar{r}_j, \mathbf{W}/\mathbf{W}_j] = \bar{z}_j) \\ \vee p(\tilde{u}[\bar{z}_j \bar{r}_j] : \bar{\theta}_j) \wedge (\tilde{s}[\bar{r}/\bar{r}_j, \mathbf{W}/\mathbf{W}_j] = \bar{z}_j) \wedge \neg \exists \mathbf{Y}. Q[\bar{r}/\bar{r}_j, \mathbf{W}/\mathbf{W}_j] \end{array} \right) \end{array} \right)$

Fig. 2. Rewrite rule SVT for *exists sometimes* properties.

$$\begin{aligned} & \vee sq(z_2 : \mathcal{Z}_{>0}, y : \mathcal{Z}_{>0}) \wedge \neg \exists x : \mathcal{Z}_{[0,20]}. (x = z_2) \\ & \vee sq(z_2 : \mathcal{Z}_{>0}, y : \mathcal{Z}_{>0}) \wedge (x : \mathcal{Z}_{[0,20]} = z_2) \wedge \neg b(z_2) \end{aligned}$$

Chan's simplification rule can be formalized by a set of *exists sometimes* properties as follows.

$$\begin{aligned} & \forall x : \mathbf{1}. y_1 : \mathbf{1} \cdots y_n : \mathbf{1}. (x = s(y_1, \dots, y_n) \rightarrow y_1 \in \mathbf{1} \wedge \cdots \wedge y_n \in \mathbf{1}) \\ & \forall x : \mathbf{1}. \exists ? y_1 : \mathbf{1} \cdots y_n : \mathbf{1}. (x = s(y_1, \dots, y_n)) \end{aligned}$$

These satisfy (1) and (3) and allow SVT to be applied.

There is no rewrite rule with introduction of local variables for *exists* properties because introducing local variables won't lead to simplification. Let $\neg \exists \mathbf{W} \bar{r}. p(\tilde{u} \tilde{s} \bar{r})$ be the negative goal. Suppose we have $\forall \tilde{u} : \bar{\sigma}. \exists \tilde{s} : \bar{\psi} \bar{r} : \bar{\omega}. p(\tilde{u} \tilde{s} \bar{r})$ and $(type(\tilde{u}) \sqsubseteq \bar{\sigma}) \wedge \mathbf{V}_{\tilde{u}} \cap (\mathbf{W} \cup \bar{r}) = \emptyset \wedge \bar{\omega} \sqsubseteq type(\bar{r})$. By introducing local variables $\bar{z} : \bar{\psi}$, the negative goal is equivalent to $\neg \exists \mathbf{W} \bar{z} \bar{r}. (p(\tilde{u} \bar{z} \bar{r}) \wedge \bar{z} = \tilde{s})$. Applying (SVT) rewrite rule with the property that $(\bar{z} = \tilde{s})$ has at most one solution, we end up with $\neg \exists \mathbf{W} \bar{z}. (\bar{z} = \tilde{s}) \vee (\bar{z} = \tilde{s}) \wedge \neg \exists \bar{r}. p(\tilde{u} \bar{z} \bar{r})$. The negative goal $\neg \exists \bar{r}. p(\tilde{u} \bar{z} \bar{r})$ can't be rewritten using (ET) because \bar{z} are not local variables in it. Thus, introducing new local variables doesn't help. Introduction of local variables is irrelevant to the miscellaneous rewrite rule as miscellaneous properties have no output parameters.

3 Digraph

The rewrite rules (ET) and (RT) are applied to negative goals that are negation of single atom and do not involve introduction of local variables. Their implementation is much easier than the other two rewrite rules and will not be considered.

The rewrite rules QVT and SVT can be applied repeatedly to extract positive information from a negative goal $\neg\exists W.G_n, \dots, G_2, G_1$. A naive implementation would repeatedly scan a conjunction of goals and check if an atom is extractable. After an atom is extracted, some local variables become global, making it necessary to check if other atoms are extractable. That would result in an inefficient implementation because most of those checks would fail.

A previously inextricable atom becomes extractable only after some of its local variables become global or some of its global variables are given a value or a smaller type. However, neither QVT nor SVT changes the type of global variables, nor will it assign any value to them. So, after an atom is extracted, it is only necessary to check those other atoms that share with the extracted atom some variables that have become global. For that reason, we use a list Φ consisting of atoms to be checked and a digraph \mathcal{D} which links each atom with the local variables it contains. The method repeatedly removes one atom from Φ and checks for its extractability until Φ becomes empty. Digraph \mathcal{D} is used in order to quickly retrieve the local variables an atom contains and the atoms containing a particular local variable. After an atom is extracted, it is moved out of the scope of the negation and the local variables it contains become global. This is done by removing the atom and the local variables from \mathcal{D} . Before the removal of the local variables, other atoms linked to them are added to Φ as their extractability need to be checked for again. Initially, every atom need to be checked.

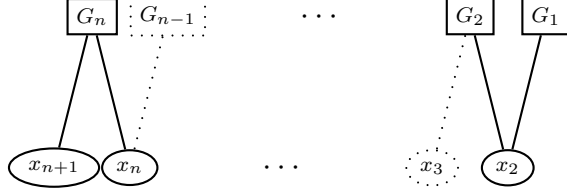
Let us first consider the case where an *existence* property has one output subtype for its output parameter. When an atom is extracted by QVT or SVT without introducing any new local variable, it is moved out of the scope of the negation and the local variables in it are promoted to being global. The atom is deleted from Φ and \mathcal{D} . The other atoms that are linked to the local variables are then added into Φ and the local variables are deleted from \mathcal{D} . The method continues with the updated \mathcal{D} and Φ .

Example 14 *Let p be of arity 2 with the following exists unique property.*

$$\forall x:\mathbf{1}.\forall y.(p(x, y) \rightarrow y \in \mathbf{1}) \quad \text{and} \quad \forall x:\mathbf{1}.\exists!y:\mathbf{1}.p(x, y)$$

Let $G_i = p(x_i, x_{i+1})$. The negative goal $\neg\exists x_2:\mathbf{1}.\dots x_{n+1}:\mathbf{1}.(G_n, \dots, G_i, \dots, G_1)$

is such that extracting G_i makes G_{i+1} extractable. A naive implementation of QVT does $\frac{n(n-1)}{2}$ tests by testing G_n for n times, G_{n-1} for $n-1$ times and so on. The negative goal has the following graph.



The proposed implementation works as follows. Initially, Φ contains G_n, \dots, G_1 that are removed from Φ and tested in that order until G_1 is extracted. At that point, only G_2 is added to Φ , it is then immediately removed and tested. Extracting G_2 adds G_3 into Φ . This process continues until G_n is tested and extracted, proving the falsity of the original negative goal. A total of $(2n-1)$ tests are performed with G_1 being tested once and each G_i for $2 \leq i \leq n$ twice.

When an atom is extracted by QVT or SVT by means of introducing local variables, only some local variables become global and the derived goals are more complex. However, the residual negative subgoals can be obtained in the same way as above.

When the output parameter of an *existence* property has more than one output subtype, several complex goals may be derived from the negative goal. Each of these complex goals may contain a number of residual negative subgoals to which QVT or SVT may be applicable. However, these residual negative subgoals differ only in the names and types of newly promoted global variables. So, the digraph and the checklist for each of these residual negative subgoals are obtained in the same way.

4 Extractability

Given an atom inside a negation and an *existence* property, QVT and SVT have to decide if the atom satisfies the *existence* property and, if so, decide for which output arguments new local variables need be introduced. The rules QVT and SVT differ only in that SVT has an extra disjunct $\neg \exists(\bar{z}_j \bar{r}_j) : \bar{\theta}_j.p(\tilde{u} \bar{z}_j \bar{r}_j)$ for each relevant output subtype. Otherwise, they are the same. The common functionality of QVT and SVT is factored out to a function *sqvt*. It tests if an atom satisfies an *existence* property, introduces new local variables, decides if an output subtype is relevant, and renames and types local variables.

```

function sqvt( $P, G, L$ )
begin
(01) Let  $G$  be  $q(\dots, t_u, \dots, t_x, \dots)$  and  $P$  be  $\langle p(\dots, \mathbf{i}(\sigma), \dots, \mathbf{o}(\bar{\Theta}), \dots), I \rangle$ ;
(02) if  $q = p$  and  $(type(t_u) \sqsubseteq \sigma) \wedge (\mathbf{V}_{t_u} \cap L) = \emptyset$  for each  $t_u$  matching an  $\mathbf{i}(\sigma)$ 
(03) then
  (04)  $\bar{r} := \epsilon; \bar{x}_m := \epsilon; \bar{z} := \epsilon; \bar{s} := \epsilon; W := nil; J := I$ ;
  (05) for each  $t_x$  matching an  $\mathbf{o}(\bar{\Theta})$  do  $J := J \cap \{k \mid (type(t_x) \sqcap \bar{\Theta}(k)) \neq \mathbf{0}\}$  od;
  (06) for each  $t_x$  at position  $p$  matching an  $\mathbf{o}(\bar{\Theta})$  do
    (07) if  $t_x \in (L \setminus \bar{r}) \wedge \forall j \in J. (\bar{\Theta}(j) \sqsubseteq type(t_x))$ 
    (08) then
      (09)  $\bar{r} := \bar{r}[p \mapsto t_x]$ ;
      (10)  $\bar{x}_m := \bar{x}_m[p \mapsto (t_x, \bar{\Theta})]$ ;
    (11) else
      (12)  $\bar{s} := \bar{s}[p \mapsto t_x]$ ;
      (13) for each  $v \in ((\mathbf{V}_{t_x} \cap L) \setminus (\bar{r} \cup W))$  do  $W := v :: W$  od;
      (14)  $z := newv(\mathbf{1}); \bar{z} := \bar{z}[p \mapsto z]; G := G[t_x/z]$ ;
      (15)  $\bar{x}_m := \bar{x}_m[p \mapsto (z, \bar{\Theta})]$ 
    (16) fi;
  (17) od;
  (18)  $\bar{x} := map(fst, \bar{x}_m)$ ;
  (19)  $\bar{x}_{cs} := \bigcup_{j \in J} \{map(\lambda e. newv((snd(e))(j)), \bar{x}_m)\}$ ;
  (20) return  $(G, \bar{x}, \bar{x}_{cs}, \bar{s}, \bar{z}, \bar{r}, W)$ 
(21) else return  $nil$ 
(22) fi
end;

```

Fig. 3. The *sqvt* function where $x :: L$ is a list with head x and tail L .

An *exists unique* property is represented as follows. Each input parameter $u:\sigma$ in $\bar{u}:\bar{\sigma}$ is represented by $\mathbf{i}(\sigma)$. Each output parameter x in \bar{x} with output subtypes $\{\theta_k \mid k \in I\}$ is represented by $\mathbf{o}(\bar{\Theta})$ where $\bar{\Theta}$ is a mapping which maps k in I to θ_k . An *exists unique* property has the following representation where input and output parameters may be interspersed.

$$\langle p(\dots, \mathbf{i}(\sigma), \dots, \mathbf{o}(\bar{\Theta}), \dots), I \rangle$$

The set of *exists unique* properties is denoted by $\Gamma_!$. We use the same representation for an *exists sometimes* property and denote the set of *exists sometimes* properties by $\Gamma_?$.

Example 15 The *exists sometimes* property in Ex. 13 is represented by this item in $\Gamma_?$: $\langle sq(\mathbf{o}(\{1 \mapsto \mathcal{Z}_{<0}, 2 \mapsto \mathcal{Z}_{>0}\}), \mathbf{i}(\mathcal{Z}_{>0})), \{1, 2\} \rangle$.

Fig. 3 defines *sqvt* with the following auxiliary functions. A call to *newv*(T) creates a new variable of type T . Given a pair, the function *fst* returns the first component while *snd* returns the second. The high order function *map*

applies a function f to a vector π point-wise: $\text{map}(f, \pi)(i) = f(\pi(i))$ for each $i \in \text{dom}(\pi)$ and $\text{dom}(\text{map}(f, \pi)) = \text{dom}(\pi)$.

Given an *existence* property P of the form $\langle p(\dots, \mathbf{i}(\sigma), \dots, \mathbf{o}(\bar{\Theta}), \dots), I \rangle$ and an atom G of the form $q(\dots, t_u, \dots, t_x, \dots)$ and a set \mathbf{L} of local variables, sqvt first checks if it is possible to replace some output arguments in G with newly introduced local variables so as to make G satisfy P . Since a new local variable can be introduced for any output argument in G , G can be made to satisfy P if $q = p$ and each of its input argument is of the type specified by P and contains no local variable. The test is done in line (02). Function sqvt returns nil from line (21) if this test fails. Otherwise, sqvt classifies every output argument according to whether a new local variable need be introduced for it or not. The variable \bar{r} holds the vector of output arguments for which no new local variables need be introduced, \tilde{s} is the vector of other output arguments and \bar{z} is the vector of corresponding newly introduced local variables. Whenever a new local variable z is introduced for an output argument t_x , sqvt substitutes z for t_x in G . The function sqvt collects the list \mathbf{W} of the local variables that occur in \tilde{s} but not in \bar{r} . It also builds up the vector \bar{x}_m of the new output arguments each of which is associated with a mapping from indices in I to types and collects the set J of relevant indices for G . Line (04) initializes these vectors and sets. Line (05) computes the set J of relevant indices. The (06)-(17) loop iterates through all output arguments. Line (07) determines if it is necessary to introduce a new local variable for the output argument t_x under consideration. If not, line (09) adds t_x into \bar{r} and line (10) adds to \bar{x}_m a pair consisting of t_x and the mapping for the corresponding output parameter in P . Otherwise, line (12) adds t_x to \tilde{s} , line (13) adds to \mathbf{W} the local variables in t_x that do not occur in \bar{r} or \mathbf{W} , line (14) introduces a new local variable z of type $\mathbf{1}$, adds z to \bar{z} and substitutes z for t_x in G , and line (15) adds to \bar{x}_m a pair consisting of z and the mapping for the corresponding output parameter in P . The newly introduced local variable z in line (14) will be renamed and attached with an appropriate type from the mapping paired with it in \bar{x}_m . Line (18) extracts the vector \bar{x} of the new output arguments of G . Line (19) makes, for each relevant index in J , a new copy of \bar{x} and types the copy with an appropriate type, and collects the set \bar{x}_{cs} of all the copies made. For a fixed index $j \in J$, line (19) does the following for each pair in \bar{x}_m . It first takes the second component of the pair which is a mapping from indices to types, then finds the type for the index j , and creates a new variable of that type. Line (20) returns with required information.

Example 16 *Continue with Ex. 15. Let $G = \text{sq}(x:\mathcal{Z}_{[0,20]}, y:\mathcal{Z}_{>0})$ and $\mathbf{L} = \{x:\mathcal{Z}_{[0,20]}\}$. Then $\text{sqvt}(P, G, \mathbf{L}) = (G', \bar{x}, \bar{x}_{cs}, \tilde{s}, \bar{z}, \bar{r}, \mathbf{W})$ with $G' = \text{sq}(z:\mathbf{1}, y:\mathcal{Z}_{>0})$, $\bar{x} = z:\mathbf{1}$, $\bar{x}_{cs} = \{z_2:\mathcal{Z}_{>0}\}$, $\tilde{s} = x:\mathcal{Z}_{[0,20]}$, $\bar{z} = z:\mathbf{1}$, $\bar{r} = \epsilon$, and $\mathbf{W} = \{x:\mathcal{Z}_{[0,20]}\}$.*

Lemma 17 *The time complexity of the test for the extractability of an atom with respect to an exists unique or exists sometimes property is linear in the*

size of the atom.

Proof. The time complexity of the function *sqt* is proportional to the size of the atom, given an atom and an existence property. When an atom is tested for its extractability, it may be necessary to match it against several different existence properties before it can be decided whether or not it is extractable. When it is not extractable, it has to be matched against all those existence properties that have the same predicate symbol as the atom. The number of the existence properties that have the same predicate symbol as an atom is bounded, which implies the time complexity of the test for the extractability of an atom is proportional to the size of the atom.

The following theorem gives the correctness of *sqt*. In addition, it states that *sqt* introduces a new variable only when it is necessary.

Theorem 18 *Let P be an exists sometimes (resp. exists unique) property, G an atom, Q a conjunction of goals and L a set of variables.*

- a) *Atom G can be extracted from $\exists L.(G \wedge Q)$ by SVT (resp. QVT) using P iff $sqt(P, G, L) \neq nil$.*

Furthermore, letting $sqt(P, G, L) = (G', \bar{x}, \bar{x}_{cs}, \tilde{s}, \bar{z}, \bar{r}, W)$,

- b) *\bar{r} , \tilde{s} , \bar{z} and W are as in SVT (resp. QVT) and \bar{r} is maximal in the sense that any proper super-vector of \bar{r} will include at least one output argument of G for which a new variable must be introduced;*
c) *$G' = G[\tilde{s}/\bar{z}]$;*
d) *\bar{x} is the vector of the output arguments of G' ; and*
e) *\bar{x}_{cs} is a set of vectors with each being a fresh copy of \bar{x} typed by an output subtype of P that is relevant to G .*

Proof. Postulate (a) follows from the conditional statement beginning at line (02). Line (05) computes the set J of relevant indices since two vectors of types with the same domain intersect iff their corresponding components at each position in the domain intersect. The logic of the loop beginning at line (06) ensures that \bar{r} , \tilde{s} and \bar{z} are computed correctly without computing their corresponding sets of indices and it also ensures the maximality of \bar{r} . Therefore, postulates (b) and (c) hold. The postulates (d) and (e) follow from lines (18) and (19) respectively.

5 Implementation

With a negative goal being represented by $neg(\Phi, \mathcal{D})$ where Φ is the checklist and \mathcal{D} is the digraph, QVT and SVT are implemented as a derivation rule

\hookrightarrow_{sqt} which derives from the lefthand side of QVT (respectively SVT) each conjunct in a disjunctive normal form of the righthand side of QVT (respectively SVT). Let $loc(\mathcal{D})$ be the set of local variables in \mathcal{D} , $delete(Ns, \mathcal{D})$ be the result of deleting nodes in Ns from \mathcal{D} , $link(N, Ns, \mathcal{D})$ be true iff \mathcal{D} links node N with some node in Ns .

- $\alpha, neg(\{G\} \cup \Phi, \mathcal{D}), \beta \hookrightarrow_{sqvt} \alpha, N_l, \beta$ for each $1 \leq l \leq k$ if $\exists P \in \Gamma_1.sqvt(P, G, loc(\mathcal{D})) = (G', \bar{x}, \bar{x}_{cs}, \tilde{s}, \bar{z}, \bar{r}, W)$ and $N_1 \vee N_2 \vee \dots \vee N_k$ is a disjunctive normal form of

$$\bigwedge_{\bar{x}' \in \bar{x}_{cs}} \left[\begin{array}{l} \text{let } \mathbf{W}' = \text{map}(\text{newv} \circ \text{type}, \mathbf{W}) \text{ in} \\ \left(\begin{array}{l} G' \wedge \neg \exists \mathbf{W}'. (\tilde{s}[\mathbf{W}/\mathbf{W}'] = \tilde{z}) [\bar{x}/\bar{x}'] \\ \vee G' \wedge (\tilde{s}[\mathbf{W}/\mathbf{W}'] = \tilde{z}) [\bar{x}/\bar{x}'] \wedge \text{neg}(\Phi', \mathcal{D}') [\mathbf{W}/\mathbf{W}'] [\bar{x}/\bar{x}'] \end{array} \right) \end{array} \right]$$

where $\Phi' = \Phi \cup \{N \mid \text{link}(N, \bar{r} \cup \mathbf{W}, \mathcal{D})\} \setminus \{G\}$ and $\mathcal{D}' = \text{delete}(\bar{r} \cup \mathbf{W} \cup \{G\}, \mathcal{D})$. The above formula corresponds to the righthand side of QVT in that \bar{x}' corresponds to $\bar{z}_j \bar{r}_j$ and \mathbf{W}' to \mathbf{W}_j . Note that \bar{x}' and \mathbf{W}' are typed when they are created.

- $\alpha, \text{neg}(\{G\} \cup \Phi, \mathcal{D}), \beta \hookrightarrow_{\text{sqt}} \alpha, N_l, \beta$ for each $1 \leq l \leq k$ if $\exists P \in \Gamma_{\gamma}.\text{sqt}(P, G, \text{loc}(\mathcal{D})) = (G', \bar{x}, \bar{x}_{cs}, \tilde{s}, \bar{z}, \bar{r}, \mathbf{W})$ and $N_1 \vee N_2 \vee \dots \vee N_k$ is a disjunctive normal form of

$$\bigwedge_{\bar{x}' \in \bar{x}_{cs}} \left[\begin{array}{l} \text{let } \mathbf{W}' = \text{map}(\text{newv} \circ \text{type}, \mathbf{W}) \text{ in} \\ \quad \neg \exists \bar{x}'. G'[\bar{x}/\bar{x}'] \\ \quad \vee G' \wedge \neg \exists \mathbf{W}'. (\tilde{s}[\mathbf{W}/\mathbf{W}' = \bar{z}])[\bar{x}/\bar{x}'] \\ \quad \vee G' \wedge (\tilde{s}[\mathbf{W}/\mathbf{W}' = \bar{z}])[\bar{x}/\bar{x}'] \wedge \text{neg}(\Phi', \mathcal{D}')[\mathbf{W}/\mathbf{W}'][\bar{x}/\bar{x}'] \end{array} \right]$$

where $\Phi' = \Phi \cup \{N \mid \text{link}(N, \bar{r} \cup W, \mathcal{D})\} \setminus \{G\}$ and $\mathcal{D}' = \text{delete}(\bar{r} \cup W \cup \{G\}, \mathcal{D})$.

- $\alpha, neg(\{G\} \cup \Phi, \mathcal{D}), \beta \hookrightarrow_{sqvt} \alpha, neg(\Phi, \mathcal{D}), \beta$
if $\forall P \in \Gamma_! \cup \Gamma_?.sqvt(P, G, loc(\mathcal{D})) = nil$. This rule removes from the checklist an atom which doesn't satisfy any *existence* property.
- $\alpha, neg(\emptyset, \Lambda), \beta \hookrightarrow_{sqvt} \text{false}$ where Λ is the empty digraph. Note that $neg(\emptyset, \Lambda)$ represents $\neg \text{true}$.

Example 19 The goal $\neg \exists x: \mathcal{R}_{[-20,20]}. u: \mathcal{R}_{\geq 0}. (sq(x, y: \mathcal{R}_{>0}) \wedge add(x, u, -1))$ is represented as F_0 below where the checklist is depicted as a group of pointers to atoms.

$$F_0 = neg(\overset{\text{sq}(x,y:\mathcal{R}_{>0})}{\text{add}(x,u,-1)})$$

Using (1') and (2'), we have $F_0 \hookrightarrow_{s_{\text{gvt}}} F_1$, $F_0 \hookrightarrow_{s_{\text{gvt}}} F_2$, $F_0 \hookrightarrow_{s_{\text{gvt}}} F_3$ and

$F_0 \hookrightarrow_{sqvt} F_4$ and $\{F_1, F_2, F_3, F_4\}$ is a frontier of F_0 where

$$\begin{aligned}
F_1 &= sq(z_1:\mathcal{R}_{<0}, y:\mathcal{R}_{>0}) \wedge \neg\exists x_1:\mathcal{R}_{[-20,20]}.(x_1 = z_1) \\
&\quad \wedge sq(z_2:\mathcal{R}_{>0}, y) \wedge \neg\exists x_2:\mathcal{R}_{[-20,20]}.(x_2 = z_2) \\
F_2 &= sq(z_1:\mathcal{R}_{<0}, y:\mathcal{R}_{>0}) \wedge \neg\exists x_1:\mathcal{R}_{[-20,20]}.(x_1 = z_1) \\
&\quad \wedge sq(z_2:\mathcal{R}_{>0}, y) \wedge (x_2:\mathcal{R}_{[-20,20]} = z_2) \wedge F_6 \\
F_3 &= sq(z_1:\mathcal{R}_{<0}, y:\mathcal{R}_{>0}) \wedge (x_1:\mathcal{R}_{[-20,20]} = z_1) \wedge F_5 \\
&\quad \wedge sq(z_2:\mathcal{R}_{>0}, y) \wedge \neg\exists x_2:\mathcal{R}_{[-20,20]}.(x_2 = z_2) \\
F_4 &= sq(z_1:\mathcal{R}_{<0}, y:\mathcal{R}_{>0}) \wedge (x_1:\mathcal{R}_{[-20,20]} = z_1) \wedge F_5 \\
&\quad \wedge sq(z_2:\mathcal{R}_{>0}, y) \wedge (x_2:\mathcal{R}_{[-20,20]} = z_2) \wedge F_6
\end{aligned}$$

with

$$\begin{aligned}
F_5 &= neg\left(\begin{array}{c} \text{add}(x_1:\mathcal{R}_{[-20,20]}, u, -1) \\ \text{u}:\mathcal{R}_{\geq 0} \end{array} \right) \\
F_6 &= neg\left(\begin{array}{c} \text{add}(x_2:\mathcal{R}_{[-20,20]}, u, -1) \\ \text{u}:\mathcal{R}_{\geq 0} \end{array} \right)
\end{aligned}$$

The following is an exists unique property for addition.

$$\forall x:\mathcal{R}.y:\mathcal{R}.z.(add(x, z, y) \rightarrow z \in \mathcal{R})$$

$$\forall x:\mathcal{R}.y:\mathcal{R}.\exists! z:\mathcal{R}.add(x, z, y)$$

Using this property, we have

$F_5 \hookrightarrow_{sqvt} add(x_1:\mathcal{R}_{[-20,20]}, v_1:\mathcal{R}, -1) \wedge \neg\exists u_1:\mathcal{R}_{\geq 0}.(u_1 = v_1)$ and $F_5 \hookrightarrow_{sqvt} add(x_1:\mathcal{R}_{[-20,20]}, v_1:\mathcal{R}, -1) \wedge (u_1:\mathcal{R}_{\geq 0} = v_1) \wedge neg(\emptyset, \Lambda)$ and $F_6 \hookrightarrow_{sqvt} add(x_2:\mathcal{R}_{[-20,20]}, v_2:\mathcal{R}, -1) \wedge \neg\exists u_2:\mathcal{R}_{\geq 0}.(u_2 = v_2)$ and $F_6 \hookrightarrow_{sqvt} add(x_2:\mathcal{R}_{[-20,20]}, v_2:\mathcal{R}, -1) \wedge (u_2:\mathcal{R}_{\geq 0} = v_2) \wedge neg(\emptyset, \Lambda)$. Since the subgoals $\neg\exists u_1:\mathcal{R}_{\geq 0}.(u_1 = v_1:\mathcal{R})$ and $\neg\exists u_2:\mathcal{R}_{\geq 0}.(u_2 = v_2:\mathcal{R})$ are equivalent to type constraints $v_1:\mathcal{R}_{<0}$ and $v_2:\mathcal{R}_{<0}$ respectively and $neg(\emptyset, \Lambda)$ is unsatisfiable, $\{add(x_1:\mathcal{R}_{[-20,20]}, v_1:\mathcal{R}_{<0}, -1)\}$ is a frontier of F_5 and $\{add(x_2:\mathcal{R}_{[-20,20]}, v_2:\mathcal{R}_{<0}, -1)\}$ is a frontier of F_6 .

$\neg\exists x_1:\mathcal{R}_{[-20,20]}.(x_1 = z_1:\mathcal{R}_{<0})$ is equivalent to type constraint $z_1:\mathcal{R}_{<(-20)}$, and $\neg\exists x_2:\mathcal{R}_{[-20,20]}.(x_2 = z_2:\mathcal{R}_{>0})$ to $z_2:\mathcal{R}_{>20}$. Solving $(x_1:\mathcal{R}_{[-20,20]} = z_1:\mathcal{R}_{<0})$ restricts the types of both x_1 and z_1 to $\mathcal{R}_{(-20,0)}$ whilst solving $(x_2:\mathcal{R}_{[-20,20]} = z_2:\mathcal{R}_{>0})$ restricts the types of both x_2 and z_2 to $\mathcal{R}_{(-20,0]}$. Therefore $\{F_7, F_8, F_9, F_{10}\}$ is a frontier of F_0 where

$$\begin{aligned}
F_7 &= sq(z_1:\mathcal{R}_{<(-20)}, y:\mathcal{R}_{>0}) \wedge sq(z_2:\mathcal{R}_{>20}, y) \\
F_8 &= sq(z_1:\mathcal{R}_{<(-20)}, y:\mathcal{R}_{>0}) \wedge sq(z_2:\mathcal{R}_{(-20,0]}, y) \wedge add(z_2, v_2:\mathcal{R}_{<0}, -1)
\end{aligned}$$

$$\begin{aligned}
F_9 &= sq(z_1:\mathcal{R}_{[-20,0]}, y:\mathcal{R}_{>0}) \wedge add(z_1, v_1:\mathcal{R}_{<0}, -1) \wedge sq(z_2:\mathcal{R}_{>20}, y:\mathcal{R}) \\
F_{10} &= sq(z_1:\mathcal{R}_{[-20,0]}, y:\mathcal{R}_{>0}) \wedge add(z_1, v_1:\mathcal{R}_{<0}, -1) \\
&\quad \wedge sq(z_2:\mathcal{R}_{(-20,0]}, y) \wedge add(z_2, v_2:\mathcal{R}_{<0}, -1)
\end{aligned}$$

Note that none of F_7, F_8, F_9 and F_{10} contain a negation!

We have implemented in ECLiPSe [1] a prototype simplification system that also implements Chan's constructive negation rule. A type is associated with a variable as an attribute [3]. The top-level of the simplification system is `neg/2`. $neg(G, L)$ is true iff $\neg\exists L.G$ is true. It constructs a digraph representation for $\neg\exists L.G$ and applies \hookrightarrow_{sqvt} repeatedly until no rewriting can be done. It then displays the derived goal.

Example 20 *This example illustrates a session with the prototype. Term $real(l, u)$ encodes type $\mathcal{R}_{[l,u]}$.*

```
[eclipse 2]: neg((sq(X:real(-0.5,0.5),U), sq(Y:real(-1,1),V),
               add(U,V,W:real(0,1))), [U,V]).
```

```
sq(Y:real(-1, 1), V1:real), add(Z:real, V1:real, W:real(0, 1)),
sq(X:real(-0.5, 0.5), U1:real), neg_eq(Z:real, U1:real, []);
```

no (more) solution.

I.e., $\neg\exists U : \mathbf{1}.V : \mathbf{1}.(sq(X:\mathcal{R}_{[-0.5,0.5]}, U), sq(Y:\mathcal{R}_{[-1,1]}, V), add(U, V, W:\mathcal{R}_{[0,1]}))$ rewrites to $sq(Y:\mathcal{R}_{[-1,1]}, V1:\mathcal{R}), add(Z:\mathcal{R}, V1, W:\mathcal{R}_{[0,1]}), sq(X:\mathcal{R}_{[-0.5,0.5]}, U1:\mathcal{R}), Z \neq U1$. The prototype incorporates existence properties of arithmetic constraints. The programmer may provide existence properties as in the following.

```
[eclipse 3]: declare_existence_property(
  eu(append(i(list(Beta)),i(list(Beta)),o([(1,list(Beta))])),[1])),
declare_existence_property(
  eu(sort(i(list(Gamma)),o([(1,list(Gamma))])),[1])),
neg((append(X:list(real),Y:list(real),Z),sort(Z,W), b(W)),[W,Z]).

append(X:list(real),Y:list(real),Z:list(real)),sort(Z,W),neg(b(W),[]).

no (more) solution.
```

Example 21 `[eclipse 1]: RGt0 = and(real(0,pinf), not(real(0,0))),`
`type_set(X,real(-20,20)), type_set(Y, RGt0),`
`type_set(U,real(0,pinf)),`
`neg([Y],(sq(X,Y),add(X,U,-1))),`
`delayed_goals(L), print(L),nl.`

```
sq(Z1:real(0,pinf) and not(real(0,20)),Y:real(0,pinf) and not(real(0,0))),
sq(Z2:real(minf,0) and not(real(-20,0)), Y:real(0,pinf) and not(real(0,0)));
```

```
sq(Z1:real(0,pinf) and not(real(0,20)), Y:real(0,pinf) and not(real(0,0))),
sq(Z2:real(-20,0) and not(real(0,0)), Y:real(0,pinf) and not(real(0,0))),
add(Z2:real(-20,0) and not(real(0,0)), V2:real and not(real(0,pinf)), -1);
```

```
sq(Z1:real(0,20) and not(real(0,0)), Y:real(0,pinf) and not(real(0,0))),
add(Z1:real(0,20) and not(real(0,0)), V1:real and not(real(0,pinf)), -1),
sq(Z2:real(minf,0) and not(real(-20,0)), Y:real(0,pinf) and not(real(0,0)))
```

```
sq(Z1:real(0,20) and not(real(0,0)), Y:real(0,pinf) and not(real(0,0))),
add(Z1:real(0,20) and not(real(0,0)), V1:real and not(real(0,pinf)), -1),
sq(Z2:real(-20,0) and not(real(0,0)), Y:real(0,pinf) and not(real(0,0))),
add(Z2:real(-20,0) and not(real(0,0)), V2:real and not(real(0,pinf)), -1)
```

no (more) solution.

6 Time Complexities

Given a negative goal, a \hookrightarrow_{sqvt} derivation step extracts an atom out of a negation and produces several residual negative goals which are then processed in subsequent derivation steps. The time complexity of \hookrightarrow_{sqvt} with respect to a negative goal is measured by the time spent on all possible derivations from the negative goal.

Our analysis is based on a notion of a spawning tree SPT_G for a negative goal G . The nodes in SPT_G are negative goals that are derived from G by repeated applications of \hookrightarrow_{sqvt} . Let G' be a node SPT_G and G'' occurs in one of the conjunctive goals derived from G' by \hookrightarrow_{sqvt} . Then G'' is a child of G' .

Let the negative goal G consist of m atoms with non-decreasing sizes $s_i, 1 \leq i \leq m$. Consider the time complexity of \hookrightarrow_{sqvt} . We weight the i^{th} atom in G by the number w_i of those atoms that share local variables with the i^{th} atom and are smaller in size than the i^{th} atom.

Some branches in SPT_G result from failed extractability tests. The parent node linked by such a branch has exactly one child and is called futile. Other nodes correspond to successful extractability tests and are called fruitful. The set of fruitful nodes in SPT_G is denoted $Fr(SPT_G)$. Let s_{nd} is the size of the atom that is extracted at a fruitful node nd and w_{nd} the weight of the atom.

Theorem 22 *Let G be a negative goal.*

- (1) *The time cost of the extractability tests performed along a path in SPT_G is $\mathcal{O}(\sum_{i=1}^m (w_i + 1) \times s_i)$.*

- (2) The time cost of all \hookrightarrow_{sqvt} derivations from G is $\mathcal{O}(\sum_{nd \in Fr(SPT_G)}(w_{nd} + 1) * s_{nd})$.

Proof. Consider (1) first. We only need to consider the worst case where each atom in G will finally be extracted. At the root, every atom in the digraph G is in the checklist. The time complexity of the extractability tests performed at the root is thus $\mathcal{O}(\sum_i s_i)$. An atom is added into the checklist only after the removal of some local variable linked to the atom. Therefore, an atom may be tested for its extractability for as many times as one plus the number of atoms with which the atom share a local variable. However, in the worst case smaller atoms are extracted before larger atoms. Thus, the i^{th} atom can only be tested for $w_i + 1$ times. Therefore, the time complexity of one derivation is $\mathcal{O}(\sum_i (w_i + 1) \times s_i)$.

Now consider (2). Since each instance of atom which is extracted at node nd is tested at most $w_{nd} + 1$ times and each test costs s_{nd} unit of time. Thus, the total cost of tests in all \hookrightarrow_{sqvt} derivations from G is $\mathcal{O}(\sum_{nd \in Fr(SPT_G)}(w_{nd} + 1) * s_{nd})$.

7 Related Work

Apart from Cleary's original work [9], most related works are those on constructive negation. The basic idea of Chan's constructive negation approach [7,8] is that answers to $\neg Q$ are obtained by negating answers to Q . Given $\neg Q$, a frontier of a derivation tree for Q is first obtained. Answers to $\neg Q$ are then obtained from the frontier as first-order formulae which are interpreted in Clark's equality theory (CET). Chan's method was formulated for logic programs in the Herbrand universe and involves introducing disequality constraints over the Herbrand universe. An answer to a goal by Chan's operational semantics SLD-CNF is a set of equality and disequality constraints. Originally, Chan's method applied only to negative goals with finite sub-derivation trees and worked by negating answers to the negated sub-goal [7]. Chan later extended his method by negating a frontier of a derivation tree for the negated sub-goal [8]. The simplification procedure in Chan's method relies on the following property of the Herbrand universe.

$$\neg \exists \bar{y}\bar{z}.(x = s(\bar{y}) \wedge Q(\bar{y}\bar{z})) \leftrightarrow \forall \bar{y}.(x \neq s(\bar{y})) \vee \exists \bar{y}.(x = s(\bar{y}) \wedge \neg \exists \bar{z}.Q(\bar{y}\bar{z}))$$

where x is a free variable and \bar{y} and \bar{z} are disjoint. Muñoz-Hernández et. al. refined Chan's method and incorporated it into Ciao Prolog [25]. They also implemented other negation methods [21] and use static analysis to select the appropriate negation method for a negative goal [26].

Małuszyński and Näslund put forward another approach to constructive negation which allows a negative goal to directly return fail substitutions, as its answers [18]. Since answers to negative goals cannot in general be represented by a finite number of substitutions, Małuszyński and Näslund’s approach sometimes need to return an infinite number of fail substitutions.

Drabent defines SL DFA resolution over the Herbrand universe [13]. Chan’s first method works only when the negated sub-goal has a finite number of answers. SL DFA overcomes this by constructing answers for the negative goal from a finite number of answers to the negated sub-goal.

Fages proposes a simple concurrent pruning mechanism over standard SLD derivation trees for constructive negation in constraint logic programs [14]. Two derivation trees are concurrently constructed. The computed answers from one of the trees are used to prune the nodes of the other. Fages’ method admits an efficient implementation as it is not necessary to deal with complex goals with explicit quantifiers outside the constraint part.

Stuckey provides a constructive negation method for constraint logic programs over arbitrary structures [29]. Stuckey’s method which is sound and complete with respect to the three-valued consequences of the completion of the program can be thought of as a generalisation of Chan’s. Stuckey uses the following property of logic formulae in his simplification procedure.

$$\neg \exists \bar{y}.(c \wedge Q) \leftrightarrow \neg \exists \bar{y}.c \vee \neg \exists \bar{y}.(c \wedge Q)$$

where c is a constraint and Q is a conjunction of goals. The method need to do a satisfiability test when combining $\neg \exists \bar{y}.c$ with other constraints. A sufficient condition for applying Stuckey’s method is that the constraint domain has the admissible closure property, i.e., $\neg \exists \bar{y}.c$ for any admissible constraint c can be rewritten as a disjunction of admissible constraints [29]. Dovier et. al. prove that the admissible closure property is also a necessary condition for an effective implementation of the method [11].

Constructive intensional negation was studied in [2,5,4,21,27]. Marchiori [19] addresses the termination of logic programs with respect to constructive negation. Lobo [17] studies constructive negation for disjunctive logic programs. Ramírez and Falaschi [28] and Moreno-Navaro [22,23,24] extend constructive negation for functional logic programs. Dovier et. al. extends Chan’s method to $CLP(SET)$ where SET is the domain of hereditarily finite sets [12]. SET does not satisfy the admissible closure property and hence the constructive negation method is complete only for a subset of $CLP(SET)$ [11].

We now compare our method with Chan’s and Stuckey’s using Ex. 11. QVT rewrites $\neg \exists z:\text{list}(\beta).(append(x:\text{list}(\beta), y:\text{list}(\beta), z), p(z))$ to $append(x:\text{list}(\beta), y:$

$\text{list}(\beta), z:\text{list}(\beta)), \neg p(z)$. Both Chan's method and Stuckey's first construct an SLD derivation tree of $\text{append}(x, y, z), p(z)$ and collect a frontier of the SLD derivation, say,

$$\left\{ \begin{array}{l} (x = [], y = z, p(z)), \\ (x = [h|x'], y = y', z = [h|z'], \text{append}(x', y', z'), p(z)) \end{array} \right\}$$

Then the negation of this frontier is simplified and put into its disjunctive normal form. This gives rise to the following four conjunctive formulae.

- (1) $x \neq [], \forall h, x'. (x \neq [h|x'])$
- (2) $x \neq [], x = [h|x'], \neg \exists z'. (\text{append}(x', y, z'), p([h|z']))$
- (3) $x = [], \forall h, x'. (x \neq [h|x']), \neg p(y)$
- (4) $x = [], x = [h|x'], \neg p(y), \neg \exists z'. (\text{append}(x', y, z'), p([h|z']))$

Stuckey's method derives (2) and (3) because the constraint parts of (1) and (4) are unsatisfiable. Chan's method derives (1), (2) and (3) as it only tests satisfiability of atomic constraints. The constraint part of (4) is failed by unification in Chan's method as $[]$ is not unifiable with $[h|x']$. Neither of these methods is effective as (2) is as complex as the original goal. The *exists unique* property allows us to obtain a simpler derived goal without making use of SLD derivation, and to eliminate unsatisfiable derived goals without satisfiability tests. Similar comparison can be made between our's and methods in [13,14,18] since they all construct a frontier of an SLD derivation tree for $\text{append}(x, y, z), p(z)$.

8 Conclusion

We have presented a simplification method that uses typed *existence* properties to rewrite negative goals. The method strictly generalizes an earlier work that uses functional dependencies to rewrite negative goals. A typed existence property generalizes a functional dependency in that the domains of both input and output parameters can be restricted to sub-domains and moreover one input value may correspond to more than one output values. The method consists of rewrite rules one for each kind of typed *existence* properties. The rewrite rules doesn't involve an SLD-derivation of the negated sub-goal nor an explicit satisfiability test.

We have described an implementation of the method and analyzed its complexity. The implementation uses a digraph and a worklist to represent a negative goal so as to avoid futile extractability tests of atoms in the negative goal. An algorithm is presented that does the extractability test given an atom and an

existence property and introduces new local variables into the atom to make it satisfy the *existence* property. The complexity of the algorithm is linear in the size of the atom.

Acknowledgement

We would like to thank anonymous referees for their constructive comments and suggestions.

References

- [1] A. Aggoun et. al. *ECLⁱPS^e 3.5 User Manual*. ECRC Munich, Germany, December 1995.
- [2] A. Bossi, M. Fabris, and M.C. Meo. A bottom-up semantics for constructive negation. In [6], pages 520–534.
- [3] P. Brisset et. al. *ECLⁱPS^e 3.4 Extensions User Manual*. ECRC Munich, Germany, July 1994.
- [4] P. Bruscoli, A. Dovier, E. Pontelli, and G. Rossi. Compiling intensional sets in CLP. In [6], pages 647–661.
- [5] P. Bruscoli, F. Levi, G. Levi, and M.C. Meo. Compilative constructive negation in constraint logic programs. *Lecture Notes in Computer Science*, 787:52–67, 1994.
- [6] M. Bruynooghe, editor. *Proceedings of the Eleventh International Conference on Logic Programming*. The MIT Press, 1994.
- [7] D. Chan. Constructive negation based on the completed database. In [16], pages 111–125.
- [8] D. Chan. An Extension of Constructive Negation and its Application in Coroutining. In *Proceedings of the 1989 North American Conference on Logic Programming*, pages 477–496. The MIT Press, 1989.
- [9] J.G. Cleary. Constructive negation of arithmetic constraints using data-flow graphs. *Constraints*, 2:131–162, 1997.
- [10] J.G. Cleary and L. Lu. Constructive negation using typed existence properties. *Lecture Notes in Computer Science*, 1490:411–426, 1998.
- [11] A. Dovier and E. Pontelli and G. Rossi. A necessary condition for Constructive Negation in Constraint Logic Programming. *Inf. Process. Lett.*, 74 (3&4):147–156, 2000.

- [12] A. Dovier, E. Pontelli and G. Rossi. Constructive Negation and Constraint Logic Programming with Sets. *New Generation Comput.*, 19 (3):209-256, 2001.
- [13] W. Drabent. What is failure? An approach to constructive negation. *Acta Informatica*, 32:27–59, 1995.
- [14] F. Fages. Constructive negation by pruning. *Journal of Logic Programming*, 32(2):85–118, 1997.
- [15] T. Frühwirth, E. Shapiro, M.Y. Vardi and E. Yardeni. Logic Programs as Types for Logic Programs. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 300-309, The IEEE Computer Society Press, 1991.
- [16] R. A. Kowalski and K. A. Bowen, editors. *Proceedings of the Fifth International Conference and Symposium on Logic Programming*. The MIT Press, 1988.
- [17] Jorge Lobo. On constructive negation for disjunctive logic programs. In *Proceedings of the 1990 North American Conference on Logic Programming*, pages 704–718, The MIT Press, 1990.
- [18] J. Małuszyński and T. Näslund. Fail Substitutions for Negation as Failure. In *Proceedings of the 1989 North American Conference on Logic Programming*, pages 461–476. The MIT Press, 1989.
- [19] E. Marchiori. On termination of general logic programs w.r.t. constructive negation. *Journal of Logic Programming*, 26(1):69–89, 1996.
- [20] A. Mycroft and R.A. O’Keefe. A Polymorphic Type System for Prolog. *Artificial Intelligence*, 23(3): 295–307, 1984.
- [21] J. Mariño and J. J. Moreno-Navarro and S. Muñoz-Hernández. Implementing Constructive Intensional Negation. *New Generation Comput.*, 27 (1):25-56, 2008.
- [22] J. J. Moreno-Navarro. Default rules: An extension of constructive negation for narrowing-based languages. In [6], pages 535–549.
- [23] J. J. Moreno-Navarro. Extending constructive negation for partial functions in lazy functional-logic languages. *Lecture Notes in Artificial Intelligence*, 1050:213–228, 1996.
- [24] J. J. Moreno-Navarro and S. Muñoz-Hernández. How to incorporate negation in a Prolog compiler. *Lecture Notes in Computer Science*, 1753:124–139, 2000.
- [25] S. Muñoz-Hernández and J.J. Moreno-Navarro. Implementation Results in Classical Constructive Negation. *Lecture Notes in Computer Science*, 3132:284-298, 2004.
- [26] S. Muñoz-Hernández, J. J. Moreno-Navarro and M. V. Hermenegildo. Efficient Negation Using Abstract Interpretation. *Lecture Notes in Computer Science*, 2250:485-494, 2001.

- [27] S. Muñoz-Hernández, J. Mariño, and J.J. Moreno-Navarro. Constructive intensional negation. *Lecture Notes in Computer Science*, 2998:39–54, 2004.
- [28] M. J. Ramírez and M. Falaschi. Conditional Narrowing with Constructive Negation. *Lecture Notes in Artificial Intelligence*, 660:59–79, 1993.
- [29] P.J. Stuckey. Negation and constraint logic programming. *Information and Computation*, 118:12–33, 1995.
- [30] E. Yardeni and E. Shapiro. A Type System for Logic Programs. *Journal of Logic Programming*, 10(2): 125–153, 1991.